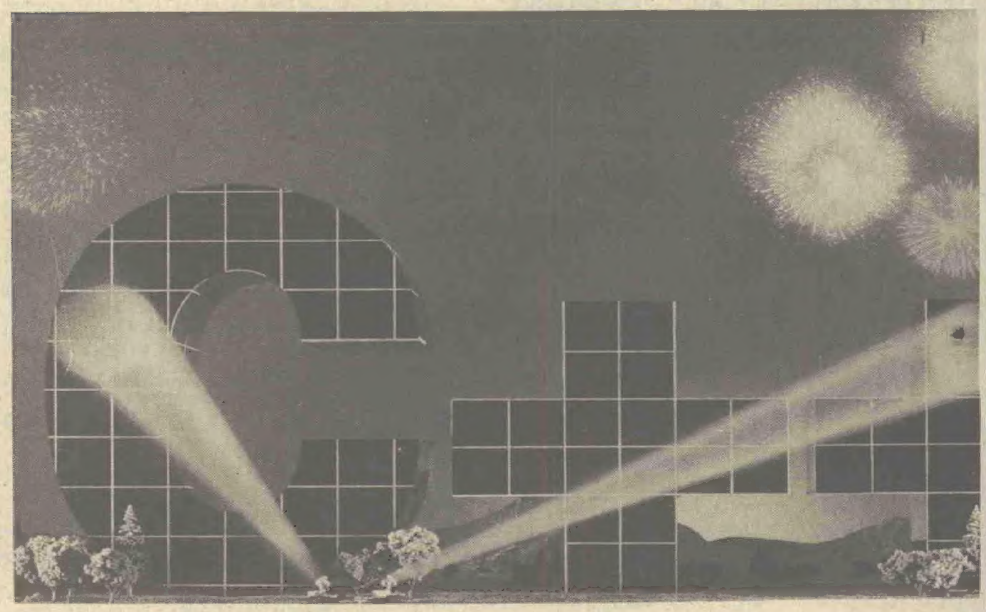
เรียนรู้การใช้งานภาษา C++ ด้วยตนเอง : ตัวดำเนินการและเฟรนด์ใน C++



ศึกษาวิธีการใช้งานตัวดำเนินการและเฟรนด์ใน C++

ฟังก์ชันสมาชิกเป็นตัวจัดการกระทำในการส่งข่าวสารตามแบบของการใช้โปรแกรมเชิงวัตถุ เมื่อฟังก์ชันสมาชิกถูกร้องขอจะมีวัตถุหนึ่งสวมบทบาทเป็นผู้รับเมจเสจ และอีกค่ากับอีกวัตถุหนึ่งจะถูกส่งผ่านทางอาร์กิวเมนต์ของฟังก์ชันด้วยเนื้อหาของข่าวสาร ดังตัวอย่างประโยค acct.deposit (100); หมายความว่าวัตถุ acct ได้รับข่าวสาร deposit ด้วยเนื้อหาของข่าวสารคือเงิน 100 บาท

ยังมีการเข้าถึงลักษณะโปรแกรมเชิงวัตถุอีกแบบหนึ่ง โดยที่วัตถุคลาสถูกจัดการด้วยฟังก์ชันและตัวดำเนินการโดยวิธีเดียวกับวัตถุข้อมูลเบื้องต้นอันได้แก่ จำนวนเต็มและทศนิยม โดยการนำเอา acct เป็นอาร์กิวเมนต์ และเอาเนื้อหาข่าวสาร (จำนวนเงินที่ต้องการฝาก) เป็นอาร์กิวเมนต์อีกตัวหนึ่งของฟังก์ชัน deposit (acct,100); ซึ่ง acct ไม่มีบทบาทพิเศษอะไร นอกจากเป็นอาร์กิวเมนต์ตัวหนึ่งในฟังก์ชัน deposit( ) เท่านั้น ถ้า acct ถูกส่งผ่านโดยวิธีการอ้างอิงแล้ว deposit( ) จะสามารถอัพเดตตัวแปรอินสแตนซ์ให้สอดคล้องกับฟังก์ชันสมาชิกได้

ฟังก์ชัน deposit( ) จะถูกโอเวอร์โหลดได้ด้วย ดังนั้นถ้าเรามีคลาสของบัญชีเงินฝากต่างกัน อาทิเช่น บัญชีเช็ค ออมทรัพย์ เดินสะพัด หรืออื่น ๆ deposit( ) ก็จะมีนิยามของแต่ละชนิดต่างกันในบัญชีเงินฝาก นั่นคือการเรียกฟังก์ชันเดียวกันจะส่งผลให้เกิดสภาวะต่าง ๆ กันตามชนิดของบัญชีเงินฝาก ดังเช่น Transfer (acct2,acct1,100) คือการโอนเงิน 100 บาทจากบัญชี acct1 ไปบัญชี acct2

วิธีการเข้าสู่โปรแกรมเชิงวัตถุวิธีที่ 2 ซึ่งเป็นการใช้กันใน C++ เรียกว่า ฟังก์ชันเฟรนด์ (friend function) มีลักษณะความเป็นเพื่อนสมชื่อ คือ สามารถจะใช้ฟังก์ชันร่วมกันได้ หรือฟังก์ชันนี้ถึงแม้ว่าจะประกาศไว้ในคลาสว่าเป็น private แต่ถ้าประกาศฟังก์ชันใหม่ให้เป็นเฟรนด์กับฟังก์ชันเก่า ฟังก์ชันใหม่ก็จะเข้าถึงฟังก์ชันเก่าได้

ฟังก์ชันเป็นเฟรนด์ของหลาย ๆ คลาสได้ แต่เป็นสมาชิกได้มากที่สุดคลาสเดียว เช่น ฟังก์ชันคอนสตรัคเจอร์ เดสตรัคเจอร์ และฟังก์ชันเวอร์ชวล (Virtual) ต้องเป็นฟังก์ชันสมาชิกเท่านั้น C++ จะบอกว่าฟังก์ชันใดจะ

เป็นฟังก์ชันสมาชิก หรือฟังก์ชันใดจะเป็นเฟรนด์ได้เอง

ในภาษาโปรแกรมส่วนใหญ่ ตัวดำเนินการจะทำหน้าที่เดียวโดยสร้างขึ้นในภาษานั้น ผู้ใช้จะนิยามฟังก์ชันใหม่ ๆ ได้เอง แต่จะไม่สามารถสร้างนิยามตัวดำเนินการเองได้ ในทางตรงกันข้าม C++ อนุญาตให้ผู้ใช้นิยามตัวดำเนินการสำหรับวัตถุคลาสได้ และตัวดำเนินการยังสามารถเป็นสมาชิกหรือเป็นเฟรนด์ของคลาสได้

การทำตัวดำเนินการโอเวอร์โหลด (Operator overload)

ทุก ๆ ตัวดำเนินการจะมีนิยามอย่างน้อยหนึ่งนิยามที่สร้างมาจากตัวภาษา และหลายตัวดำเนินการมีการโอเวอร์โหลดมาแล้ว

การนิยามตัวดำเนินการที่มีมากกว่า 1 ชนิด เช่น ตัวดำเนินการ + อาจจะนิยามเป็นตัวดำเนินการของข้อมูลชนิดจำนวนเต็ม (int) long, float, double และอื่น ๆ ใน C++ จะไม่ยุ่งกับนิยามของตัวดำเนินการที่สร้างโดยภาษาดั้งเดิมเลย (คือไม่อนุญาตให้ผู้ใช้สร้างนิยามของตัวดำเนินการที่ได้นิยามไว้ในตัวภาษาแล้วเช่น int,long, flort)

เราพยายามจัดนิยามสำหรับตัวดำเนินการ โดยนิยามฟังก์ชันที่มีชื่อว่า operator ตามด้วยสัญลักษณ์ของตัวดำเนินการ เช่น

เราจัดให้มีนิยามสำหรับตัวดำเนินการ + โดยนิยามฟังก์ชัน operator +

เราจัดให้มีนิยามสำหรับตัวดำเนินการ = โดยนิยามฟังก์ชัน operator =

ตัวดำเนินการผลลัพธ์ << น่าจะเป็นตัวอย่างที่ดีของตัวดำเนินการโอเวอร์โหลด ตามปกติการนิยม << ที่สร้างขึ้นโดย << เป็นตัวดำเนินการเลื่อนบิตของค่าตัวจำนวนเต็ม ไฟล์เฮดเดอร์ iostream.h ประกาศคลาส ostream ของสตรีมผลลัพธ์และจัดให้มีจำนวนของนิยาม << ซึ่งตัวดำเนินการตัวแรกเป็นวัตถุ ostream ซึ่งนิยามนี้ทำให้เราใช้ << เป็นผลลัพธ์ของค่าหลายชนิดเช่น long, double, char (สตริง) และรวมถึงชนิดของข้อมูลที่ผู้ใช้นิยามขึ้นใช้เอง

ซึ่งในตัวดำเนินการข้อมูลเข้า >> ก็มีลักษณะคล้าย ๆ กัน การประกาศของ << สำหรับผลลัพธ์ของค่า int จะมีรูปแบบดังนี้

อาร์กิวเมนต์ของฟังก์ชันจะสอดคล้องกับโอเปอร์แรนด์ของตัวดำเนินการ ดังนั้นตัวดำเนินการ << รับเอาสตรีมผลลัพธ์เป็นโอเปอร์แรนด์ ตัวแรกและจำนวนเต็ม (int) เป็นผลลัพธ์ตัวที่ 2 สตรีมผลลัพธ์ถูกส่งผ่านโดยการอ้างอิงเพราะภาวะภายในของมันจะถูกดัดแปลงด้วยตัวดำเนินการผลลัพธ์

ตัวดำเนินการ << จะคืนค่าอ้างอิงให้แก่ โอเปอร์แรนด์ตัวแรกด้วยสตรีมผลลัพธ์ c คุณสมบัตินี้ทำให้เราสามารถใช้ << หลาย ๆ ตัวดำเนินการเรียงกันได้ เช่น

```
cout << 100 << "\n";
```

มีความหมายเหมือนกับ

```
(cout << 100) << "\n";
```

การคำนวณจะทำในวงเล็บก่อน คือการส่ง 100 ไปที่ cout แล้วส่งคืนค่าแบบอ้างอิงให้แก่ cout การอ้างอิงนี้รับหน้าที่คล้ายกับโอเปอร์แรนด์ตัวแรกสำหรับตัวดำเนินการ << ตัวที่ 2 จะทำหน้าที่เหมือนกับ cout << "\n";

นิพจน์ cout << "\n" จะคืนค่าแบบอ้างอิงให้แก่ cout ดังนั้นตัวมันเองจึงถูกตามด้วยตัวดำเนินการ << ตัวอื่นได้เรื่อย ๆ

ส่วนการประกาศของ << สำหรับผลลัพธ์ที่เป็นทศนิยม (double) มีรูปแบบดังนี้

```
ostream & operator <<
(ostream & c, int n);
```

นิยามที่แตกต่างออกไปจะเกิดขึ้นตามแต่ละการประกาศ ชนิดของโอเปอร์แรนด์จะเป็นตัวตัดสินว่านิยามไหนจะเป็นตัวที่ถูกใช้ ด้วยประโยคคำสั่ง

```
cout << 100;
```

เป็นการบอกว่าผลลัพธ์ที่ออกมานั้นมีค่าเป็นจำนวนเต็ม

```
cout << 3.5
```

เป็นการบอกว่าผลลัพธ์ที่ได้มีค่าออกมาเป็นทศนิยม ซึ่งไม่มีทศนิยมของโปรแกรมส่งผลต่อทศนิยมดั้งเดิมของตัวภาษาเลยถ้าโอเปอร์แรนด์เป็นจำนวนเต็มทั้งคู่ เช่น 100 << 5 ก็ทำหน้าที่เป็นตัวดำเนินการแบบบิตเหมือนเดิม

เป็นการง่ายที่เดียวที่จะนิยามตัวดำเนินผลลัพธ์สำหรับสิ่งที่เราทำการประดิษฐ์ขึ้นมาเอง เช่น สมมติเรานิยามคลาส clock__time ด้วย

```
struct clock__time {
        int hr;
        int min;
        int sec;
};
```

ถ้าเราประกาศตัวแปรของ clock__time ด้วย t ดังนี้

```
clock__time t = { 11, 45, 20 }
```

แสดงว่าเราต้องการให้ cout << t; พิมพ์ผลลัพธ์ออกจอภาพเป็น 11:45:20

เราสามารถนิยามตัวดำเนินการผลลัพธ์สำหรับวัตถุ clock__time ดังต่อไปนี้

```
ostream & operator <<
(osstream & c, clock__time t)
{
c << t.hr << ":";
c << t.min << ":";
c << t.sec ;
return c;
}
```

มีจุดสำคัญหลายจุดที่สมควรหรือมีคุณค่าที่จะบันทึกไว้ในที่นี้คือ

● มันเป็นสิ่งที่ประหลาดที่ตัวดำเนินการ << ถูกนำมาใช้ในนิยามของ << อย่างไรก็ตาม ตัวดำเนินการ << ในนิยามของผลลัพธ์ที่เป็นจำนวนเต็มและสตริง เรานำเอานิยามของ << ที่มีอยู่แล้วเรียกกลับมาใช้ใหม่ โดยที่โอเปอร์แรนด์ตัวที่ 2 จัดเป็นจำนวนเต็มหรือสตริง

● เพราะว่า clock__time เป็น struct ตัวแปรสมาชิกของมันเป็น public ซึ่งสามารถที่จะถูกอ้างถึงอย่างอิสระในนิยามตัวดำเนินการ ถ้าตัวแปรอินสแตนซ์เป็นไปรเวท ตัวดำเนินการจำเป็นต้องถูกประกาศเป็นเฟรนด์ของคลาส clock__time เพื่อที่จะได้เข้าถึงตัวแปรสมาชิกได้

● ตามที่ได้อธิบายมาเรียบร้อยแล้ว << คืนค่าแบบอ้างอิงแก่โอเปอร์แรนด์ตัวแรกและตัวดำเนินการ << หลายตัวสามารถที่จะเขียนเรียงกันได้ ผู้ใช้สามารถจะสร้างนิยามสำหรับตัวดำเนินการใน C++ ได้ยกเว้น กับตัวต่อไปนี้

```
::
. ตัวดำเนินการเข้าถึงสมาชิก
.*
sizeof ตัวดำเนินการขนาด
? : ตัวดำเนินการนิพจน์เงื่อนไข
```

สิ่งที่เราโอเวอร์โหลดได้คือ ตัวดำเนินการขอร้องจากฟังก์ชัน, ( ), ตัวดำเนินการ ตัวดัชนี [ ] ตัวดำเนินการเปลี่ยนชนิดข้อมูล เช่น (int) ตัวดำเนินการกำหนดค่า, = และตัวดำเนินการจัดการหน่วยความจำ เช่น new และ delete

ถึงแม้ว่าเราจะขยายความหมายของตัวดำเนินการออกไปด้วยการสร้างนิยามขึ้นใหม่ แต่เราก็ไม่เปลี่ยนแปลงไวยากรณ์ (syntax) โดยเฉพาะอย่างยิ่งเราไม่สามารถเปลี่ยนแปลงคุณสมบัติต่อไปนี้ของตัวดำเนินการได้ เช่น

arity จำนวนตัวดำเนินการ

การทำก่อน (precedence) ตัดสินว่าตัวดำเนินการจะถูกทำให้เป็นกลุ่มเพื่อจะสร้างนิพจน์ย่อยได้หรือไม่

การจัดกลุ่ม ตัดสินว่าตัวดำเนินการที่มีการทำก่อนที่เหมือนกันจะอยู่รวมเป็นกลุ่มจากซ้ายไปขวาได้หรือไม่

ตัวดำเนินโอเวอร์โหลดจะเป็นตัวดำเนินเอกภาพ (คือมีโอเปอร์แรนด์ตัวเดียว) หรือไบนารี (โอเปอร์แรนด์ 2 ตัว) ก็ได้ แต่ถ้าเป็นตัวดำเนินการแบบไทรนารี (trinary) จะไม่สามารถโอเวอร์โหลดได้ เช่น ?:

ตัวดำเนินการ 3 ตัวคือ + - และ * ใช้เป็นตัวดำเนินการเอกภาพหรือไบนารีก็ได้ นั่นคือจะให้เป็นนิยามทั้งตัวดำเนินการเอกภาพและไบนารีของแต่ละอันได้ ส่วนตัวดำเนินการตัวอื่นจะเป็นเอกภาพหรือไบนารีได้เพียงอย่างใดอย่างหนึ่ง และนิยามของตัวดำเนินการจะต้องให้สอดคล้องกับจำนวนโอเปอร์แรนด์ด้วย

เราทราบมาก่อนแล้วว่าตัวดำเนินการเอกภาพ ++ และ -- ใช้ได้ทั้งเป็นตัวดำเนินการเติมหน้า (++i,;--i) และเติมหลัง (i++,i--) โดยการใช้นิยามที่สร้างขึ้นด้วยตัวภาษาต่าง ๆ กัน สำหรับการใช้ตัวดำเนินการเติมหน้าหรือเติมหลังไม่จำเป็นที่ผู้ใช้จะต้องสร้างนิยามขึ้นมาใหม่เอง

เมื่อฟังก์ชันตัวดำเนินการ อาทิเช่น operator+ ไม่เป็นฟังก์ชันสมาชิกของคลาส จำนวนอาร์กิวเมนต์ที่ประกาศสำหรับฟังก์ชันเหมือนกันกับจำนวนโอเปอร์แรนด์ และต้องสอดคล้องกับตัวดำเนินการนั้น เช่น ตัวดำเนินการ << ต้องการ 2 โอเปอร์แรนด์ จึงนิยาม operand<< ให้ไม่เป็นสมาชิกของคลาสใด ๆ ดังนั้นต้องประกาศให้มี 2 โอเปอร์แรนด์ ดังนี้

```
ostream & operator <<
(ostream & c, clock__time t);
```

เราคงจำได้ว่าฟังก์ชันสมาชิกถูกจัดให้เป็นวัตถุคลาสเฉพาะได้ด้วยการวางไว้ก่อนตัวดำเนินการจุด เช่น

```
acct.deposit (100);
```

แต่สำหรับตัวดำเนินการสมาชิกวัตถุคลาสนี้มักจะปรากฏอยู่เป็นโอเปอร์แรนด์ตัวแรกเสมอ ดังนั้นจำนวนอาร์กิวเมนต์ที่ประกาศสำหรับฟังก์ชัน operator จะมีน้อยกว่าจำนวนโอเปอร์แรนด์อยู่หนึ่ง เพราะว่าโอเปอร์แรนด์ตัว

แรกเป็นวัตถุคลาสเสมอที่ซึ่งตัวดำเนินการจัดหามาให้ เช่น

สมมติว่าเราต้องการนิยามตัวดำเนินการ += ให้เป็นตัวดำเนินการฝากเงิน สำหรับคลาส account หรือว่า acct += 100; จะแทนการฝากเงิน 100 บาท ในบัญชี acct การประกาศ operator += เป็นสมาชิกแบบ public ของ account เราอาจจะสอดแทรกการประกาศต่อไปนี้ในส่วนที่เป็น public ของคลาส account ก็ได้เช่น

```
double operator += (double amt);
```

ข้อสังเกต จะเห็นว่าอาร์กิวเมนต์ตัวที่สองของ operator+= ประกาศอย่างเห็นได้ชัดว่า อาร์กิวเมนต์ตัวแรกจะเป็นวัตถุคลาสที่ตัวดำเนินการจัดหามาได้ ดังนั้นเราจึงนิยาม operator += เหมือนกับฟังก์ชันสมาชิกของ account ดังแสดงในรูปที่ 1

```
double account :: operator += (double amt)
{
balance += amt; // uses built-in definition of += return amt;
}
```

รูปที่ 1

ฟังก์ชันเฟรนด์และตัวดำเนินการ (Friend Functions and Operators)

นิยามของฟังก์ชันหรือตัวดำเนินการจะสามารถประกาศให้เป็นเฟรนด์ของคลาสใด ๆ ก็ได้ นิยามนั้นจะมีการเข้าถึงส่วนที่เป็นไปรเวทของคลาสได้ โดยทำให้เป็นเฟรนด์ หรือกล่าวอีกนัยหนึ่งก็คือ มันไม่ได้เข้าถึงอย่างมีการยอมรับนัก โดยการเป็นฟังก์ชันสมาชิก เช่น acctl.bolance เพราะ acct2.rate จะอ้างอิงสมาชิกของคลาสที่มีชื่อเป็นเฟรนด์ได้

ตัวอย่าง สมมติว่าเราต้องการนิยามฟังก์ชัน transfer( ) ซึ่งโอนจำนวนเงินระหว่าง 2 บัญชี ฟังก์ชันนี้น่าจะทำให้เป็นเฟรนด์ เพราะว่ามันส่งผลแก่บัญชี 2 บัญชี มิใช่บัญชีเดียว เราจึงประกาศว่า transfer เป็น friend ของ account ด้วยการใส่เข้าไปในส่วนที่เป็น public ของ account ดังนี้

```
friend double transfer (account & acct1,
                        account & acct2,
                        double amt);
```

จะเห็นว่าอาร์กิวเมนต์ account ส่งผ่านด้วยการอ้างอิง เพื่อที่วัตถุจะได้ใช้เป็นอาร์กิวเมนต์ actual (ไม่ได้ทำสำเนาเก็บไว้) เพราะง่ายต่อการเปลี่ยนแปลง ฟังก์ชันข้างต้นโอนจำนวนเงินจาก acct2 ไปยัง acct1 และคืนผลลัพธ์เป็นจำนวนเงินที่มีอยู่ปัจจุบันออกมา

```
double transfer (account & acct1,
                 account & acct2,
                 double amt)
{
if (amt <= acct2.balance) {
acct2.balance -= amt;
acct1.balance += amt;
return amt;
}
else
return 0;
}
```

ข้อสังเกต ถ้าฟังก์ชันเกี่ยวข้องกับวัตถุ account มากกว่าหนึ่ง เช่น acct1.balance และ acct2.balance จำเป็นมากในการอ้างถึงตัวแปรอินสแตนซ์

นิยามตัวดำเนินการถูกประกาศเป็นเฟรนด์ในวิถีทางเดียวกัน เช่น เราสร้างคลาส account ด้วยตัวดำเนินการผลลัพธ์ซึ่งพิมพ์ยอดเงินในบัญชีพร้อมอัตราดอกเบี้ยสำหรับวัตถุ account เราใส่การประกาศต่อไปนี้ในส่วนที่เป็น public ของคลาส account

```
friend ostream & operator << (ostream & c,
                              account a);
```

นิยามของตัวดำเนินการนี้จะแปรเปลี่ยนไปขึ้นอยู่กับเราต้องการพิมพ์ข้อมูล account อย่างไร วิธีหนึ่งที่เป็นไปได้จะแสดงในรูปที่ 2

อัตราดอกเบี้ยที่ถูกคูณด้วย 100 เพื่อจะเปลี่ยนให้อยู่ในรูปของร้อยละ ถ้าวัตถุ acct ถูกประกาศโดย

```
account acct (666.6, 0.5);
```

แล้ว cout << acct ; จะพิมพ์ข้อความ

```
balance : 666.6 rate : 0.5
```

เมื่อไรที่ควรจะประกาศฟังก์ชันหรือตัวดำเนินการเป็นสมาชิกของคลาส

```
ostream & operator << (ostream & c, account a)
{
c << "balance" : "<< a.balance << " ";
c << "rate"    : "<< 100.0 * a.rate;"
return c;
}
```

รูปที่ 2

และเมื่อไรจะประกาศเป็นเฟรนด์ ถ้าการร้องขอฟังก์ชันทำให้เกิดการเปลี่ยนแปลงภาวะภายในของวัตถุคลาสเดียวละก็ ถ้าเราคิดได้ว่าการร้องขอเป็นการส่งข่าวสารไปให้แก่วัตถุแล้ว ฟังก์ชันควรจะเป็นสมาชิกของคลาส แต่ถ้าฟังก์ชันนั้นมีผลต่อหลาย ๆ วัตถุ ซึ่งอาจจะเป็นของคลาสต่าง ๆ กันแล้ว ฟังก์ชันนั้นควรจะเป็นเฟรนด์ของคลาสแน่นอนอย่างไม่ต้องสงสัย

การเป็นสมาชิกและการเป็นเฟรนด์สามารถจะรวมเข้าด้วยกัน สมาชิกของคลาสหนึ่งสามารถจะเป็นเฟรนด์ของคลาสอื่นได้ ตัวอย่างเช่น อาร์กิวเมนต์ของฟังก์ชันสมาชิกอาจเป็นวัตถุคลาสของคลาสอื่น ถ้านิยามฟังก์ชันต้องการเข้าถึงตัวแปรอินสแตนซ์ของอาร์กิวเมนต์ ฟังก์ชันสมาชิกต้องถูกประกาศเป็นเฟรนด์โดยให้สอดคล้องกับคลาสที่มีอยู่เดิม

เราสามารถจะประกาศฟังก์ชันสมาชิกของคลาสหนึ่งเป็นเฟรนด์ของคลาสอื่นได้ เช่น สมมติมีคลาสชื่อ account_manager โดยที่ฟังก์ชันสมาชิกต้องการเข้าถึงตัวแปรอินสแตนซ์ของวัตถุ account เราก็ใส่ประกาศ

```
friend class account_manager;
```

เข้าไปในคลาส account แล้วประกาศทุก ๆ ฟังก์ชันสมาชิกของ account_manager เป็นเฟรนด์ของ account ตัวดำเนินการซึ่งไม่มี side effects (ไม่เปลี่ยนตัวโอเปอร์แรนด์) เป็นการดี ถ้าประกาศเป็นเฟรนด์ ตัวดำเนินการที่มี side effect ต่อโอเปอร์แรนด์ตัวแรกควรจะประกาศเป็นสมาชิกของคลาส

คลาสเวกเตอร์ (Class vector)

เวกเตอร์ คือ ปริมาณที่ทั้งขนาดและทิศทาง เช่น ความเร็ว ความเร่ง การเคลื่อนที่ เวกเตอร์ใช้กันมากในวิชาคณิตศาสตร์ ฟิสิกส์ และคอมพิวเตอร์ โดยเฉพาะคอมพิวเตอร์กราฟิก การควบคุมหุ่นยนต์ คลาสเวกเตอร์ของเวกเตอร์จะเป็นตัวอย่างหนึ่งในตอนนี้

ในทางคณิตศาสตร์เวกเตอร์คือชุดของจำนวน จำนวนในเวกเตอร์หนึ่งเวกเตอร์จะมี 3 จำนวน ถ้าต้องการเพิ่มหรือลดก็ทำได้โดยง่าย แต่ไฟล์ต้นฉบับต้องคอมพายล์ใหม่ถ้าจำนวนนั้นเปลี่ยนแปลงไป เราจะนิยามการบวก การลบ การคูณ สำหรับเวกเตอร์ การบวกหรือลบเวกเตอร์ก็เพียงนำเอาจำนวนในวงเล็บที่สมนัยกันบวกหรือลบกัน เช่น

```
(1, 2, 3) + (4, 5, 6) = (5, 7, 9)
(1, 2, 3) - (6, 5, 4) = (-5, -3, -1)
```

ในที่นี้เครื่องหมาย = ใช้ในวิชาคณิตศาสตร์ ไม่ใช่ใน C++

เราทำให้เวกเตอร์เป็นลบได้ด้วยการเปลี่ยนจำนวนทุกตัวเป็นตรงกันข้าม เช่น

```
-(1, 2, 3) = (-1, -2, -3)
```

ซึ่งทั้งตัวดำเนินการเอกภาพและไบนารีจะนิยามใหม่เพื่อไว้ใช้ในเวกเตอร์

ในทางคณิตศาสตร์ จำนวนเพียงจำนวนเดียวเรียกว่าสเกล่าร์ (scalar) ในการคูณสเกล่าร์กับเวกเตอร์เราเพียงคูณจำนวนทุกตัวด้วยเลขจำนวนนั้น เช่น

```
2*(1, 2, 3) = (2, 4, 6)
```

หรือ

```
(1, 2, 3)*3 = (3, 6, 9)
```

รูปแบบอื่นของการคูณเรียกว่าผลคูณสเกล่าร์ คือการคูณ 2 เวกเตอร์เข้าด้วยกัน แล้วเอาผลที่ได้รวมกัน ผลนี้จะเป็นจำนวนจำนวนหนึ่งเรียกว่าสเกล่าร์ เช่น

```
(1, 2, 3) * (4, 5, 6) = 1*4 + 2*5 + 3*6
                      = 4 + 10 + 18
                      = 32
```

เวกเตอร์ที่แทนขนาดและทิศทาง วิธีทั่วไปก็คือสนใจสเกล่าร์จากเวกเตอร์ โดยดูเฉพาะขนาดโดยไม่สนใจทิศทาง เช่น เวกเตอร์แทนการเคลื่อนที่ ขนาดจะแทนระยะทางที่เคลื่อนที่ไป ถ้าเวกเตอร์แทนความเร็ว ขนาดก็จะแทนความเร็ว

ขนาดของเวกเตอร์คำนวณได้จากการเอาจำนวนในวงเล็บยกกำลังสองบวกเข้าด้วยกัน แล้วถอดรากที่สอง

```
ขนาดของ (1, 2, 3)
        = 1*1+2*2+3*3
        = 1+4+9
        = 14
        = 3.74166
```

ในการนิยามคลาสเวกเตอร์ จุดหมายของเราคือการหาเวกเตอร์มีลักษณะที่เราคุ้นเคยกันในทางคณิตศาสตร์ เช่น ต้องการจะใช้เครื่องหมาย + แทนการบวกเวกเตอร์ เครื่องหมาย – แทนการลบเวกเตอร์ และแทนนิเสธของเวกเตอร์ * แทนการคูณเวกเตอร์ทั้งสองแบบตามที่กล่าวมาแล้ว การหาขนาดของเวกเตอร์ก็จะใช้วิธีการเปลี่ยนชนิดของข้อมูล

ถ้าเวกเตอร์ถูกใช้ในนามของสเกล่าร์ เวกเตอร์ก็จะถูกเปลี่ยนเป็นขนาดโดยอัตโนมัติ

คลาสเวกเตอร์ยังจะทำเป็นตัวดำเนินการตัวดัชนี ซึ่งจะอ้างถึงจำนวนแต่ละจำนวนของเวกเตอร์ได้

ลองสร้างคลาสเวกเตอร์ โดย List1 ในรูปที่ 3 จะเป็นไฟล์เฮดเดอร์ List2 จะเป็นไฟล์ต้นฉบับ ซึ่ง Vector.h บางทีก็มีอยู่แล้วใน C++ เราจึงเปลี่ยนชื่อใหม่เป็น Vectorc.h และ Vectorc.cpp เพื่อจะเป็นไฟล์เฮดเดอร์และไฟล์ต้นฉบับของเรานั่นเอง

รูปที่ 3

List1

```
// File vetorc.h
// header file for class vector
# imclude < iostream.h >
const Lower = 1; // Lower bound for [ ] operator
const Upper = 3; // Upper bound for [ ] opreator
const size = upper - lower + 1; // number of components
class vector {
    double c [SIZE]; // array of components
 public:
   vector ( );       // create null vector
   vector (double * a ); //create vector from array
// Addibion,subtraction,and multiplication operators
  friend vector operator + (const vector & v,
                            const vector & w);
  friend vector operator - (const vector & v,
                            const vector & w);
  friend vector operator - (const vector & v);
  friend vector operator * (double a, const vector & w);
  friend vector operator * (const vector v,double b);
  friend bouble operator * (const vector & v,
                            const vector & w);
//conparison operators
  friend int operator == (const vector & v,
                          const vector & w);
  friend int operator != (const vector & v,
                          const vector & w);
//operations conbined with assignment
 vector & operator + = (const vector & w );
 vector & operator - = (const vector & w );
 vector & operator * = (double b);
 double & operator [ ] (int i) ; // subseripting
 operator donble ( ) const; // convert to maguitude
//Insertion and extraction operators
 friend istream & operator >> (istream & ci, vector & v);
 friend ostream & operator << (ostream & co;
                               const vector & v);
```

คอนสตรัคเจอร์ (Constructors)

มีการประกาศใช้คอนสตรัคเจอร์สำหรับคลาส vector อยู่ 2 ที่ คือ ที่แรก vector( ); เป็นตัวสร้างเวกเตอร์ที่มีจำนวนตัวเลขทุกตัวเป็นศูนย์ ดังนั้น vector v; จะสร้างเวกเตอร์ชื่อ v โดยมีจำนวนตัวเลขทุกตัวเป็นศูนย์ ที่ที่สอง คือ vector (double*a);

อนุญาตให้ v เวกเตอร์มีค่าเริ่มต้นที่ถูกชี้ไปที่อะเรย์ด้วยอาร์กิวเมนต์ a ตัวอย่างเช่น ถ้าเราประกาศอะเรย์ b โดย

```
double b[SIZE] = {10, 20, 30};
```

แล้ว

```
vector v = b;
```

คือการประกาศ v ว่าเป็นเวกเตอร์ที่มีจำนวนตัวเลขเป็น 10, 20, 30 หรือจะพูดอย่างสั้นว่าการมีคอนสตรัคเจอร์ตัวที่สองเป็นการอนุญาตให้อะเรย์ถูกใช้เป็นเวกเตอร์คงตัว ในการประกาศอะเรย์เช่นนั้น ทำให้ผู้เขียนโปรแกรมมั่นใจว่าขนาดของอะเรย์จะเท่ากับจำนวนสมาชิก

ถ้ามองดูนิยามของทั้งสองคอนสตรัคเจอร์ใน List2 จะเห็นว่านิยามคอนสตรัคเจอร์ตัวแรกจะเก็บค่าศูนย์ในสมาชิกของ C และคอนสตรัคเจอร์ตัวที่สองจะทำสำเนาสมาชิกของอะเรย์อาร์กิวเมนต์แล้วส่งไปยังสมาชิกที่สมนัยกันของ C

กำหนดค่าเริ่มต้นและการกำหนดค่าใหม่ให้แก่สมาชิก

เวกเตอร์ที่ประกาศไว้แล้วสามารถที่จะกำหนดค่าเริ่มต้นด้วยค่าของเวกเตอร์ที่มีอยู่แล้วได้ และค่าของเวกเตอร์หนึ่งจะถูกกำหนดให้เป็นค่าของอีกเวกเตอร์ได้ เช่น

```
vector v = w;
v = w;
```

ถ้าเราต้องการจะเก็บเอาตัวดำเนินการอันเก่าไว้ เราต้องนิยาม copy constructor (ซึ่งเป็นการกำหนดค่าให้แก่คลาสวัตถุคลาสใหม่ จากที่มีอยู่แล้ว) และตัวดำเนินการ การกำหนดค่าให้แก่สมาชิก = : ดังแสดงในรูปที่ 4

ใน C++ การผ่านอาร์กิวเมนต์หรือการคืนค่าผลลัพธ์ด้วยการใช้ค่าก็คือการปฏิบัติการกำหนดค่าเริ่มต้น ตัวดำเนินการทั้งคู่จะทำการเตือนให้ copy constructor ให้กำหนดค่าเริ่มต้นให้แก่ตัวสำเนาตัวใหม่ให้เป็นค่าที่ถูกส่งผ่านมาหรือค่าที่ให้ผลลัพธ์คืนกลับไป ดังนั้นอาร์กิวเมนต์สำหรับ copy constructor ทำตามที่เราได้นิยามไว้ด้วยการเป็นโอเปอเรเตอร์สมาชิก โอเปอร์แรนด์ตัวแรกของ = จะเป็นวัตถุคลาสที่โอเปอเรเตอร์ถูกประยุกต์ใช้งาน และโอเปอร์แรนด์ตัวที่สองจะถูกประกาศเป็นอาร์กิวเมนต์เท่านั้น โอเปอร์แรนด์ที่สมนัยกับอาร์กิวเมนต์ w จะคอยเตรียมค่าที่ถูกทำสำเนาไว้ หรือเตรียมค่าที่ถูกกำหนดค่าเอาไว้ และ const เป็นคีย์เวิร์ดที่ประกันว่าค่าของโอเปอร์แรนด์ตัวนี้จะไม่ถูกเปลี่ยนแปลง โดยการใช้ copy constructor หรือโอเปอร์แรนด์กำหนดค่า

```
vector (const vector & w); // copy constructor
vector & operator = (const vector & w); //assignment
```

รูปที่ 4

ถ้าโอเปอเรเตอร์สองตัวคือ copy constructor หรือ assignment โอเปอเรเตอร์ไม่ได้ถูกนิยามไว้แล้วโดยปริยาย โอเปอเรชันที่สมนัยกันจะถูกนำมาใช้มีชื่อว่า การกำหนดค่าเริ่มต้นสมาชิก และการกำหนดค่าให้แก่สมาชิก ซึ่งต่อไปจะเรียกว่า memberwise initialization and assignment ตัวแปรอินสแตนซ์ที่อยู่ทางซ้ายของเครื่องหมายเท่ากับจะถูกกำหนดค่าหรือกำหนดให้ค่าของตัวแปรอินสแตนซ์ที่สมนัยกันของวัตถุที่อยู่ทางขวาของเครื่องหมายเท่ากับ

ตัวอย่าง ถ้า u และ v เป็นเวกเตอร์ การกำหนดค่า v ให้แก่ u จะเขียนได้ดังนี้

```
u = v;
```

การนิยามค่าคงที่ (Constant Definitions)

ไฟล์เฮดเดอร์เริ่มด้วยชุดของการนิยามค่าคงที่ ซึ่งหาจำนวน Components ในหนึ่งเวกเตอร์ และวิธีการที่จะให้ค่าของจำนวนในแต่ละ Components ซึ่งสามารถจะปรับปรุงคลาสให้เข้ากับการประยุกต์ โดยเพียงแต่เปลี่ยนนิยามแล้วคอมพายล์ไฟล์ต้นฉบับใหม่เท่านั้นเอง

LOWER เป็นตัวดัชนีของ Components แรกของเวกเตอร์

UPPER เป็นตัวดัชนีของ Components สุดท้ายของเวกเตอร์

นั่นคือตัวดัชนีของ Components ที่เรียงจาก LOWER กับ UPPER

ค่าคงที่ SIZE ที่ถูกนิยามในพจน์ของ LOWER กับ UPPER บอกจำนวนของ Components

ดังนั้นใน List1 เวกเตอร์ ๆ หนึ่งจะประกอบด้วย 3 Components ซึ่งจะเป็น Components ตัวที่ 1 ถึง ตัวที่ 3

```
const lower = 1;
const upper = 3;
const size = upper-lower+1;
```

The Component Array

เราเลือก double เป็นชนิดข้อมูลแบบสเกล่าร์ และเป็น Components ของเวกเตอร์ Components ของเวกเตอร์อยู่ในอะเรย์ C ซึ่งมีจำนวนสมาชิกเท่ากับ SIZE และเป็นข้อมูลชนิด double

```
double C [SIZE]
```

สมาชิกของ C เป็นตัวแปรอินสแตนซ์ของวัตถุเวกเตอร์

ข้อสังเกตุ C เป็นอะเรย์ธรรมดาใน C++ ที่ตัวดัชนีจะมีค่าตั้งแต่ศูนย์ถึง SIZE-1 พิสัยของตัวดัชนีนิยามด้วย lower และ upper ใช้เมื่อตอนทำการโอเวอร์โหลด การดำเนินตัวดัชนีซึ่งถูกประยุกต์ให้เป็นวัตถุเวกเตอร์ ถึงแม้ว่าสมาชิกของ C เป็นตัวแปรอินสแตนซ์สำหรับวัตถุเวกเตอร์ แต่ตัว C เองไม่ใช่วัตถุเวกเตอร์เป็นเพียงอะเรย์ใน C++ ธรรมดาเท่านั้น